PROGRAM EXECUTION METHOD

BACKGROUND OF THE INVENTION:

1. Field of the Invention

The present invention relates to a program execution method for dynamically compiling a program that is created using a programming language, such as Java.

2. Description of the Related Art

With the growth in popularity of Java, the importance of providing an environment that is specially suited for the use of a dynamic compiler has increased. For an execution environment for Java in which a dynamic compiler is used, if all program optimization processing, for which an extended period of time is required, is performed at the time a program is initially compiled, depending on the compile time, a reduced processing speed will be attained by the program when it is executed. Therefore, for the initial running of a program, an interpreter is employed, or compiled code is used that was produced rapidly and for which no optimization processing was performed. At this time, data is acquired concerning the execution of the program, such as the execution count and information concerning the performance, including the number of iterations performed by each loop, and this data is subsequently used for optimization. By proceeding in this manner, the rapid execution of a program can be implemented.

A conventional, well known compiling method is one called adaptive compilation, as described in "Optimizing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches," Urs Hoelzle, Craig Chambers and David Ungar, in ECOOP '91 Conference Proceedings, pp.21-38, Geneva, Switzerland, July 1991, published as Springer Verlag LNCS 512, 1991 [HCU91].

According to this method, compiling is performed substantially without optimization before a program is run the first time, and an invocation counter is employed to count the times the program is activated. Thereafter, when the activation count for a program exceeds a threshold value, an optimization count is incremented and recompiling is performed.

Since with this method a recompiled program is executed only at the time at which the pertinent method is called, the conventional method is not effective for a program that, even though it is activated only once, includes a loop or loops for which iterations are repeatedly performed, and that for its execution requires an extended period of time (includes a loop to be optimized, and a loop for which multiple iterations are performed).

A conventional method called deoptimization is also well known, as described in "Debugging Optimized Code With Dynamic Deoptimization," Urs Hoelzle, Craig Chambers and David Ungar, in Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation, pp. 21-38, June 1992 [HCU92].

According to this method, unoptimized code that is created to debug code for an optimized method, and to perform the debugging of the execution program, is transferred to the obtained code. With this method, (1) before compiling is performed for optimization, interrupt points are designated at two positions: at the beginning of the method and at a branch command near the end of a loop; (2) optimization compiling is performed with an additional restriction that optimization that exceeds an interrupt point will not be performed; (3) when an interrupt is issued during program execution, the execution of the program is resumed following each interrupt point; and (4) a stack frame for the interrupt points is stored. Then, (5) a method is compiled without being optimized; and (6) the stack frame is reproduced in consonance with the non-optimized state, and the process enters an interrupt state. This method is similar to the present invention for transferring the process during the execution of a program. However, since a transfer target is un-optimized code, and since the execution condition, such as a register image, at the transfer point can be uniquely determined using source code, this

conventional method can not be applied directly for a transfer to optimized code for which the process at the transfer destination becomes complex.

Further, a conventional well known method is called dynamic recompilation, as described in "Optimizing Dynamical-Dispatched Calls With Run-Time Type Feedback," Urs Hoelzle and David Ungar, in Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation, pp. 326-336, published as SIGPLAN Notices 29(6), June 1994 [HU94].

This method is used to provide an explanation and an overview of a method whereby, during the execution of a non-optimized method, the pertinent method is replaced by one that was optimized when it was compiled. According to this method, (1) a method to be recompiled is detected, (2) a mark is set at a point at which execution is to be resumed, and (3) values that are pointed to by all the currently effective registers are calculated. When these processes have been successfully performed, (4) a non-optimized method in a stack is replaced by an optimized one. If such a replacement is disabled, generated code is called by a succeeding method and is employed. While detailed processing has not been explained for the above method, the following problems have arisen because, as it is described, the processing is the reverse of that performed in deoptimization [HCU92].

The problems with the above method are that a plurality of transfer points are not handled; that optimization passing beyond a transfer point is not performed; and that an increase in memory is not taken into account.

Two methods are available to switch the execution of a program when optimization compiling is performed: a method whereby an execution process is changed by calling the next method, and a method whereby a method that is being executed is replaced by optimized code. The second method, especially, is a function required to increase the processing speed for a method that includes a loop for which multiple iterations are performed, even though it is activated only once, and that for its execution requires an extended period of time. However, when the

conventional method, whereby the methods are switched during execution, is employed, the following two problems are encountered.

First, optimization does not pass beyond a position (hereinafter referred to a transfer point) in a method that is to be switched. Therefore, if a transfer point is present, the quality of code obtained is lower than the code that is generated when there is no transfer point. The execution speed for a program when a succeeding method is called is lower than the execution speed obtained when optimization compiling is performed while no transfer point is present.

Especially when a transfer point has been inserted into a loop, optimization of the loop, which can greatly affect the execution performance, can not be performed, and as a result, considerable degradation of the performance occurs.

Second, no consideration is given to a case in a multithreaded environment in which processes are replaced at a plurality of different transfer points, and compiling is performed each time a transfer occurs. These multiple compiling sessions contribute to the deterioration of the execution speed; generated codes overlap and the consumption of memory is increased.

SUMMARY OF THE INVENTION

To resolve these shortcomings, it is one object of the present invention to provide a program execution method whereby higher optimization that pass beyond a transfer point can be performed, even when in a program a change is made at the transfer point.

It is another object of the present invention to provide a program execution method whereby, even when processes at multiple transfer points in a multithreaded environment are transferred, multiple processing sessions for compiling are not required, and generated codes do not overlap.

To achieve the above objects, according to the present invention, a program execution method, for transferring, from an interpreter process to a compiled code process, a method that is currently being executed for code that includes a plurality of transfer points, comprises the steps of: moving the transfer points for code to the top of a loop process when no problem occurs, even when the transfer points are moved to the top of the loop process; copying to a location immediately preceding the loop process, when the transfer points are located inside the loop process, a point that post-dominates the top of the loop process and the transfer points; storing information for generating recalculation code for specific transfer points when the moving of the code and privatization and a common sub-expression elimination that are performed pass beyond the specific transfer points; and performing a recalculation during a transfer process.

Preferably, the program execution method further comprises a step of: defining as a new transfer point, a point from the interpreter process to the compiled code process whereat, when the method that is currently being executed is replaced, the execution speed is increased compared with when the execution is not transferred.

Preferably, the program execution method further comprises the steps of: generating information required to perform a transfer from the interpreter process to the compiled code process; and storing the obtained information while correlating the obtained information with the transfer points, wherein, at the recalculation step, the information stored for the transfer points is employed.

According to the present invention, a program for transferring, from an interpreter process to a compiled code process, a method that is currently being executed for code that includes a plurality of transfer points, comprises the steps of: moving the transfer points for code to the top of a loop process when no problem occurs, even when the transfer points are moved to the top of the loop process; copying to a location immediately preceding the loop process, when the transfer points are located inside the loop process, a point that post-dominates the top of the loop process and the transfer points; storing information for generating recalculation code

for specific transfer points when the moving of the code and privatization and a common sub-expression elimination that are performed pass beyond the specific transfer points; and performing a recalculation during a transfer process.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a diagram showing the configuration of a program execution apparatus according to the present invention.

Fig. 2 is a first flowchart showing the processing performed by the program execution apparatus according to the present invention.

Fig. 3 is a second flowchart showing the processing performed by the program execution apparatus according to the present invention.

Fig. 4 is a third flowchart showing the processing performed by the program execution apparatus according to the present invention.

Fig. 5 is a fourth flowchart showing the processing performed by the program execution apparatus according to the present invention.

Fig. 6 is a fifth flowchart showing the processing performed by the program execution apparatus according to the present invention.

Fig. 7 is a sixth flowchart showing the processing performed by the program execution apparatus according to the present invention.

Fig. 8 is a seventh flowchart showing the processing performed by the program execution apparatus according to the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENT

The essential points of the present invention will now be described.

Method for preventing deterioration of code quality due to the presence of a transfer point

When a transfer point is present, there are two reasons the quality of code is deteriorated. One is that many loop optimization processes can not be performed when transfer points are present in a loop. The other is that an optimization method, such as the moving of code, and privatization or common sub-expression elimination, for storing a state in the middle of a calculation in a register or a local area in a memory, can not be performed. The following methods are employed to resolve these problems.

1. If no execution problem occurs, even when a transfer point is moved to the top of a loop, the transfer point in compiled code is moved to the top of a loop.

2. When the transfer point is located inside the loop, the process from the top of the loop to a point that post-dominates the top of the loop and the transfer point is copied to a location immediately preceding the top of the loop, and the loop is so modified that the transfer point is located outside the loop or at the top of the loop. As a result, the presence of the transfer point does not adversely affect the optimization of the loop.

3. When the movement of code and the privatization of the common sub-expression elimination are performed above the transfer point, information for generating recalculated code for the pertinent transfer point is stored at the transfer point. During a transfer process, recalculation is performed.

Method for employing code to cope with a normal call and a transfer in the middle of the execution of a program

A point that is to be transferred from the interpreter, and a point that will be transferred as a result of examining a method of a program and that will, as a result of the transfer, increase the processing speed are detected, and are defined as transfer points.

For each transfer point, transfer information, consisting of the register use state at a current transfer, point and recalculation code information are generated. In consonance with the addresses of the original code, a table including the transfer information is generated and is positioned at a location that the method can access. Upon the execution of the transfer process, the table is accessed by using the address of a transfer point as a key, and the transfer information is obtained.

There is one method whereby an unneeded variable area is not generated when a stack frame used by an interpreter is changed to a stack frame that is used by compiled code. According to this method, in the interpreter all the local variables that are designated by the individual methods are allocated to stack frames. However, in the compiled code, a restoration location is shared with a plurality of local variables, or a variable is provided that is stored only in a register and its value is not restored in the stack frame. Therefore, not all the local variables designated by the methods are allocated to the stack frames. To resolve this problem, the following procedure is employed.

A local variable area, in a stack frame that compiled code requires, is stored at a location that a method can access during a compiled code process. The size of this allocated area is no larger than the size required by compiled code. The transfer information includes information concerning whether a variable at a transfer point is stored in the register or in the stack frame. During the transfer process, first, a stack frame that the interpreter is using is restored, and a stack frame for compiled code is generated. Then, the transfer information is employed to copy the values of required local variables to registers or to the stack frame. Finally, the stack frame of the interpreter is removed, and the execution process is shifted to the transfer point.

When the interpreter process is changed only once to the execution of machine code, whereby the dynamic compiler is activated, buffer code is generated in a stack area to prevent an increase in the memory consumed by the buffer code.

The processing for the present invention will now be described.

Case wherein during the execution process an interpreter determines a transfer is required

When an interpreter determines during the execution process that a transfer is required, the following processing is performed.

1. If a method has already been compiled, the method is used to prepare a transfer information table to acquire transfer information from the address for a command at a transfer point. When the transfer information can be obtained, process 2 under "Interpreter process" below and the following processes are performed. When the transfer information can not be obtained, the transfer process is halted and the interpreter continues the execution processing.

2. A dynamic compiler is activated. At this time, the address of a transfer determination command and the address of a command to be executed after the transfer has been performed are transmitted to the compiler. When the address of a command to be executed after the transfer has been performed is obvious, this process may be eliminated.

Processing performed by a dynamic compiler

The following process is performed by a dynamic compiler.

1. The code for the method is divided to obtain a Basic Block (hereinafter simply referred to as a BB) and a control flow graph is generated. A transfer point designated by the interpreter is defined as the first in the BB.

2. A mark that includes the transfer point designated by the interpreter is provided for the BB.

3. The code for the method is examined to obtain a proposed transfer point, other than that designated by the interpreter, that can be transferred and that will provide a noticeable effect when the transfer is performed. This process is not performed when the defined transfer point is the point that was designated by the interpreter. The proposed transfer point is obtained by removing, from all the positions in the method that can be detected by the interpreter, locations at which the compiler can determine that no transfer effects will be provided. A point at which no transfer effect will be provided is, for example, a point outside a loop.

4. If no execution problem is encountered, even when the transfer point is moved to the top of the loop, the transfer point in the compiled code is moved to the top of the loop. A case where a problem in a transfer is encountered is one where a write command to a heap area is present, or one where a specific variable is defined between the top of a loop and the transfer point, and a reference for the variable is located between the two and the value of the variable can not be recalculated at the top of the loop. When recalculation can be performed, the recalculation information for the transfer point are generated.

5. If the transfer point is located inside the loop, the loop is modified so that the transfer point serves as the starting point for the loop (a loop to be optimized, including a multiplexed loop). As a result, the presence of the transfer point has no adverse affect on optimization. For this modification, code is copied from the entry point of the loop to the point in the loop that post-dominates the entry point and the transfer point. In a

worst case, the copied code is smaller than the code for the loop. Further, when the point to be transferred by the interpreter is limited to a branch command toward the end of a loop, only the control code for the loop need be copied in a Java "for" loop, while no code need be copied for a "do-while" loop.

6. When a process for moving code, or caching values using a common sub-expression or privatization, is performed that passes beyond a transfer point, the data for generating the recalculation code is stored for all the transfer points that are passed.

7. Code consonant with the method is generated.

8. The local variable area used by the compiled code is stored at a location that the method can access.

9. For each transfer point, transfer information are generated that include the code address of the transfer point, recalculation code data for caching, and register employment information for a transfer point. A transfer information table is prepared that is paired with the address of a transfer command in the original code. The table is then stored at a location that the method can access. The transfer information may be represented as actual code, or may be represented as data to be transmitted to the common transfer method. If the interpreter is changed to the execution of machine code only one time, whereat the dynamic compiler is activated, the transfer information can be generated in the stack area, so that the data can be immediately eliminated when the transfer has been completed.

Interpreter process

The following processes are performed by the interpreter:

1. When the dynamic compiling was successful, the transfer information is obtained from the transfer data table. When the compiling failed, the current transfer process is halted and program execution using the interpreter is continued.

2. The local variable area used for compiled code is obtained, necessary information in the stack frame that was employed by the interpreter is restored, and a stack frame for the compiled code is generated.

3. Necessary information is entered in the stack frame for the code that has been compiled using the transfer information. When the transfer information is represented as code, the code is executed. When the transfer information is represented as data, the data are transmitted to the transfer process method to begin the transfer processing. The transfer processing can be implemented by (1) copying, from a restoring area, a value to be stored in the stack area; (2) recalculating the value using the recalculation information, and entering the result in the local variable area, the register or the restoring area; and (3) loading the value into a register.

4. Program execution is begun at the address of the transfer point.

Program execution apparatus 1

Fig. 1 is a diagram illustrating the arrangement of a program execution apparatus 1 according to the present invention.

As is shown in Fig. 1, in the program execution apparatus 1 a server 10 and a client 12 are connected via a network 106.

The server 10 comprises Java source code 100, a Java bytecode compiler (JAVAC) 102 and Java bytecode 104.

The client 12 includes Java programs 120 and 130, which are received from a storage medium 140 and which are executed via a storage device 14 by hardware 132.

The Java program 120 includes a Java bytecode verifier 122, a Java interpreter 124, a JIT compiler 126 and a native code execution block 128.

In the server 10, the Java source code 100 is converted into bytecode 104 by a bytecode compiler, such as the JAVAC 102, and the obtained bytecode is transmitted via the network 106 to the client 12.

In the client 12, the bytecode verifier 122 verifies the bytecode 104 received from the server 10.

The Java interpreter 124 executes the verified bytecode 104 if it is not compiled by the JIT compiler 126.

The JIT compiler 126 compiles the verified bytecode 104 and generates native code.

The native code execution block 128 executes native code that is generated as a result of the compiled code process.

An explanation will now be given for one embodiment wherein the present invention is applied for the Java interpreter 124 and the JIT compiler 126.

The conditions governing the employment of this embodiment are as follows.

The Java interpreter 124 performs a transfer only upon receipt of a command that instructs a branch toward the end of a loop (a loop to be optimized, including a multiplex loop). A flag is set for a command that determines there will be no transfer, and a transfer is prohibited at a succeeding Thread.

It should be noted that after a transfer has been completed, program execution is resumed when a command is issued at the destination for the rearward branch performed in accordance with the command that initiated the transfer.

The Java interpreter 124 does not handle a transfer point at the transfer destination address, but upon the receipt of the rearward branching command that initiated the transfer.

In addition to the rearward branching command, issued when the Java interpreter 124 that activated the JIT compiler 126 determined that the transfer should be effected, the JIT compiler 126 detects, as a transfer point, another source for a rearward branching command for which a flag has not been set and that has not been optimized.

A loop surrounding a transfer point is not to be optimized, and thus the quantity of code that must be copied in order to move the transfer point outside the loop is reduced. This limitation is appropriate for the following reasons. If the surrounding loop is a "for" loop, the transfer is not currently performed because the Java interpreter 124 must determine the condition of the loop by permitting it to perform at least one iteration. Thus, the pertinent loop is assumed to be a loop having few iterations.

The transfer information is then generated as execution code.

To simplify the explanation, it should be noted that, in the code generated by the JIT compiler 126, the local variable area included in the original program is always allocated for a stack frame.

The following terms will be employed in this embodiment.

Transfer point: a point whereat program execution is transferred from the Java interpreter 124 to the JIT compiler 126.

Transfer bwd jump: a rearward branching command used by the Java interpreter 124 to detect a transfer point.

Transfer bwd jump address: an address in a memory wherein a bytecode transfer bwd jump command for a currently executing method is stored.

Transfer point table: a table wherein a transfer bwd jump address and the entry address for pad code are entered as a pair.

Method information (hereinafter simply referred to as mb): a Java structure in which method information is included.

Method compile information (hereinafter simply referred to as minfo): a structure for storing, during a compiled code process, various information concerning a method.

Process performed by the Java interpreter 124 (process 1)

The Java interpreter 124 detects a backedge by which it is immediately transferred to the JIT compiler 126.

If necessary, a value cached in the register is written in the memory.

The JIT compiler 126 is called to compile source code. At this time, the address of the backedge, which triggered the transfer, is transmitted to the JIT compiler 126.

Process performed by the JIT compiler 126 (process 2)

Locking is performed to provide an exclusion process for the compiling.

If the compiling has already been completed when the transfer bwd jump address is received, the transfer point table is examined. When the pertinent transfer point address has been included in the table, the table offset is transmitted as a return value (end).

When the transfer bwd jump address is received, in the method information a flag (hereinafter referred to as a gen_tp flag) is set to represent the execution of a process for the transfer point.

The rearward branches included in the loop are counted, and the transfer point table is allocated.

The bytecode is traversed to build a basic block (hereinafter referred to as a BB) and a loop. When the above flag is set, the following processing is performed.

The BB of the transfer point, which is obtained from the transfer bwd jump data received from the Java interpreter 124, is written in the method information.

A flag that represents the transfer point is set in the BB. The BB wherein this flag is set should not merge with the preceding BB.

At a rearward branch for forming the loop, a command for which a flag indicating no transfer is not set is entered in the transfer point table.

When a transfer point is included in the loop, the following processing is performed.

In accordance with the control flow graph, the code from the top of the loop to the transfer point is traced. During the tracing, whether the next command is present is determined. When the command is not present, the transfer point is moved to the top of the loop.

That is, in accordance with the control flow graph, the code from the top of the loop to the transfer point is traced.

Then, during the tracing process, whether the next command is present is determined. When the command is not present, the transfer point is moved to the top of the loop, and a command for writing data to the heap area is executed to end the processing for moving outside the loop.

Following this, a command for defining the local variable is executed.

Thereafter, the code that extends from the top of the loop to the transfer point is copied to a location outside the loop, and the transfer point is also moved outside.

The common sub-expression elimination process is performed.

When the process is extended across the transfer point, and when the calculation to be removed can be performed by a recalculation using a local variable belonging to the Java interpreter 124, the local variable is registered as recalculation information at the transfer point. When the recalculation can not be performed, the extension of the process across the transfer point is prohibited.

The privatization is performed for the field variable.

When the process is extended across the transfer point, and when the calculation to be removed can be performed by a recalculation using the local variable belonging to the Java interpreter 124, the local variable is registered as recalculation information at the transfer point. When the recalculation can not be performed, the extension of the process across the transfer point is prohibited.

The following processing is performed for code generation.

By using the information contained in the transfer point table for "minfo," a transfer point table wherein code (original code address, transfer code) is entered as an element and a table sized

storage area is prepared in the location that can be accessed by the mb. The bytecode address of the transfer point is then written.

The bytecode addresses are rearranged in the ascending order to facilitate the search.

The information contained in the transfer point table and the register image of the transfer point are employed to generate recalculation code for caching and a transfer code for each transfer point (code for storing a value in memory in a register). The generated codes are written in the transfer point table entered as transfer point code addresses.

A work area (a local variable area and a stack area) required by the JIT compiler 126 is established in a location that can be accessed by the mb.

Process performed by the Java interpreter 124 (process 3)

The return value for the compiler is examined. When the compilation failed, the execution of the Java interpreter 124 is continued (End).

The transfer point table for the mb is obtained to determine whether the table contains a current bwd jump address. When there is no address in the table, the execution of the Java interpreter 124 is continued. (End).

While taking into account the size of a work area received by the JIT compiler 126, the stack frame used for the Java interpreter 124 is converted into one for the JIT compiler 126.

The transfer point table is obtained from the mb.

The transfer point table is obtained, and the transfer point code address is also acquired from the address for the bytecode that is currently being executed.

The program execution is started at the transfer point code address.

The processing performed by the program execution apparatus 1 of the invention will now be described while referring to the flowcharts in Figs. 2 to 8.

The present invention is positioned as the process at S14 in the bytecode execution process (S10 in Fig. 2).

As is shown in Fig. 2, at step 100 (S100), the program execution apparatus 1 executes a method.

At step 102 (S102), the program execution apparatus 1 determines whether the method should be executed by the interpreter. When the method is to be executed by the interpreter, program control advances to S104. In the other case, program control is shifted to S114.

At step 104 (S104), the program execution apparatus 1 begins the processing using the interpreter.

At step 106 (S106), the program execution apparatus 1 reads a command.

At step 108 (S108), the program execution apparatus 1 determines whether compiled code is to be executed. When the code is to be executed, program control advances to S16 in S14. In the other case, program control is shifted to S110.

At step 110 (S110), the program execution apparatus 1 uses the interpreter to execute the code.

At step 112 (S112), the program execution apparatus 1 terminates the process that was using the interpreter.

At step 114 (S114), the program execution apparatus 1 uses the compiler to compile the method.

At step 116 (S116), the program execution apparatus 1 executes the native code that the compiler generated.

Fig. 3 is a flowchart showing the process at S14 according to the present invention.

As is shown in Fig. 3, at step 160 (S160) during the preprocessing (S16) for a transfer that is included in S14, the program execution apparatus 1 determines whether the method has been compiled. If the method has been compiled, program control advances to S162. In the other case, program control is shifted to S20, which will be described later while referring to Figs. 4 to 8.

At step 162 (S162), the program execution apparatus 1 obtains the transfer information table from the method.

At step 164 (S164), the program execution apparatus 1 examines the transfer data table, while using as a key the command address for the transfer point.

At step 142 (S142), the program execution apparatus 1 determines whether the transfer information is present. If the transfer information is present, program control advances to the transfer process (S18). In the other case, program control is shifted to S146.

At step 144 (S144), the program execution apparatus 1 determines whether the compiling performed at S20 was successful. If the compiling was performed successfully, program control advances to S18. In the other case, program control is shifted to S146.

At step 146 (S146), the program execution apparatus 1 returns to the execution process for which the interpreter is used.

At step 148 (S148), the program execution apparatus 1 terminates the processing.

At step 180 (S180) of the transfer process (S18), the program execution apparatus 1 obtains the transfer information from the transfer information table.

At step 182 (S182), the program execution apparatus 1 restores necessary information in a current stack frame.

At step 184 (S184), the program execution apparatus 1 changes the size of the stack frame by using the transfer information.

At step 186 (S186), the program execution apparatus 1 generates a stack frame for native code by using the transfer information and the restoring information.

At step 188 (S188), the program execution apparatus 1 obtains a transfer address by using the transfer information, and begins execution of the program. After the execution of the program has been completed, the program execution apparatus 1 terminates the processing (S148).

Fig. 4 is a flowchart showing the compiled code process (S20) for a transfer shown in Fig. 3.

At step 202 (S202) in S20, the program execution apparatus 1 divides the code of the method into basic blocks.

At step 204 (S204), the program execution apparatus 1 determines whether a transfer point is the head of the basic block. If the transfer point is the head, program control advances to S208. In the other case, program control is shifted to S206.

At step 206 (S206), the program execution apparatus 1 divides the basic block so that the transfer point is located at its head.

At step 208 (S208), the program execution apparatus 1 generates a control flow graph.

At step 210 (S210), the program execution apparatus 1 sets a mark in the basic block headed by the transfer point, and advances to the processing for searching for another proposed transfer point (S22 in Fig. 5).

Fig. 5 is a flowchart showing the processing (S22) performed in Fig. 4 when another proposed transfer point is searched for.

As is shown in Fig. 5, at step 222 (S222) in S22, the program execution apparatus 1 initiates an iteration process for code other than the transfer point in the method.

At step 224 (S224), the program execution apparatus 1 determines whether the process may be transferred from the interpreter. If such an action is not possible, program control advances to S226. In the other case, program control is shifted to S236.

At step 226 (S226), the controlling program analyzes the need for a transfer.

At step 228 (S228), the program execution apparatus 1 determines whether noticeable transfer effects can be provided. If noticeable effects can be provided by the transfer, program control advances to S230. In the other case, program control is shifted to S232.

At step 230 (S230), the program execution apparatus 1 determines whether the transfer point is located at the head of the basic block. If the transfer point is located at the head, program control advances to S234. In the other case, program control is shifted to S232.

At step 232 (S232), the program execution apparatus 1 divides the basic block so that the transfer point is located at the head of the basic block, and corrects the control flow graph.

At step 234 (S234), the program execution apparatus 1 sets a mark in the basic block of the transfer point.

At step 236 (S236), the program execution apparatus 1 terminates the iteration process for code other than the transfer point in the method.

At step 238 (S238), the program execution apparatus 1 terminates the processing at S22, and performs the processing (S26 in Fig. 6) for moving the transfer point outside the loop, without modifying the loop.

SUB A6

Fig. 6 is a flowchart showing the processing (S26) used to move the transfer point outside the loop, without modifying the loop in Fig. 4.

As is shown in Fig. 6, at step 262 (S262) in S26, for each transfer point the program execution apparatus 1 repeats up to S300 the processing for loop 1.

At step 264 (S264), the program execution apparatus 1 determines whether the transfer point is located in a loop that can be optimized. If the transfer point is located in such a loop, program control advances to S266. In the other case, program control is shifted to S300.

At step 266 (S266), the program execution apparatus 1 determines whether there is a writing command in the loop for an area that can be referred to by a component other than the method. If such a writing command is present in the loop, program control is shifted to S300. In the other case, program control advances to S268.

At step 268 (S268), the program execution apparatus 1 calculates a set Vr of local variables whose values, which are defined outside the loop, are referred to in the loop.

At step 270 (S270), the program execution apparatus 1 calculates a set Vd of local variables that are defined in the loop.

At step 272 (S272), the program execution apparatus 1 repeats the processing for loop 2 up to S282 for each variable VI in a set (Vr 3 Vd) of local variables that require recalculation.

At step 274 (S274), the program execution apparatus 1 collects information required for a recalculation of the variable VI at the loop start point.

At step 276 (S276), the program execution apparatus 1 determines whether recalculation of the variable VI can be performed at the loop start point. If recalculation of the variable VI can be performed, program control advances to S278. In the other case, program control is shifted to S300.

At step 278 (S278), the program execution apparatus 1 calculates a set Vc of local variables that are required for a recalculation of the variable VI.

At step 280 (S280), the program execution apparatus 1 determines whether the set (Vc 3 Vd) is empty. If the set (Vc 3 Vd) is empty, program control advances to S282. In the other case, program control is shifted to S300.

At step 284 (S284), the program execution apparatus 1 moves the transfer point to the loop start point.

At step 286 (S286), the program execution apparatus 1 determines whether the set (Vr 3 Vd) is empty. If the set (Vr 3 Vd) is empty, program control advances to S300. In the other case, program control is shifted to S288.

At step 288 (S288), the program execution apparatus 1 generates one basic block.

At step 290 (S290), the program execution apparatus 1 generates a route for a control flow that extends from the newly generated basic block to the loop start point.

At step 292 (S292), the program execution apparatus 1 changes the route leading from outside the loop to the loop start point into a route that leads to the newly generated basic block.

At step 294 (S294), the program execution apparatus 1 repeats for recalculation of each local variable the processing extending from loop 3 up to S298.

At step 296 (S296), the program execution apparatus 1 generates recalculation code using the recalculation information, and adds the obtained code to the code in the newly generated basic block.

At step 302 (S302), by modifying the loop, the program execution apparatus 1 advances to the processing (S32 in Fig. 7) for moving the transfer point to a position outside the loop.

Fig. 7 is a flowchart showing the processing (S32) for using the loop modification shown in Fig. 4 to move the transfer point outside the loop.

As is shown in Fig. 7, at step 322 (S322) in S32, for each transfer point the program execution apparatus 1 repeats the processing for loop 1 up to S234.

At step 324 (S324), the program execution apparatus 1 determines whether the transfer point is located in a loop that can be optimized. If the transfer point is located in such a loop, program control advances to S286. In the other case, program control is shifted to S334.

At step 326 (S326), the program execution apparatus 1 obtains a basic block BBp in a loop that immediately post-dominates the loop start point and the transfer point.

At step 328 (S328), the program execution apparatus 1 copies a control flow graph that is formed using the sum of the routes from the loop start point to the basic block BBp.

At step 330 (S330), the program execution apparatus 1 changes all the control flows that extend from outside the loop to the loop start point into control flows in a flow graph copy that extends to the basic block that corresponds to the loop start point.

At step 332 (S332), the program execution apparatus 1 maintains the control flows extending from the original control flow graph to the basic block BBp.

At step 336 (S336), the program execution apparatus 1 returns to S208 in S20 in Fig. 4.

An explanation for the processing will be given by referring again to Fig. 4.

At step 208 (S208), while taking a transfer point into account, the program execution apparatus 1 performs redundancy elimination.

At step 210 (S210), the program execution apparatus 1 performs optimization for which a transfer point need not be taken into account.

At step 212 (S212), the program execution apparatus 1 generates native code for the method, and advances to the processing (S34 in Fig. 8) for generating transfer information.

Fig. 8 is a flowchart showing the processing (S34) for generating transfer information in Fig. 4.

At step 342 (S342) in S34, the program execution apparatus 1 stores a stack frame, having the size that is required by the compiled code, at a location that can be accessed by the method.

At step 344 (S344), the program execution apparatus 1 repeats for each transfer point the processing performed up to S328 for loop 1.

At step 346 (S346), the program execution apparatus 1 generates transfer information (the code address of a transfer point, cache recalculation information obtained by redundancy

elimination, register use state information for a transfer point, and the address of a command by which the interpreter determines whether a transfer is to be made).

At step 350 (S350), the program execution apparatus 1 returns to S144 in Fig. 3.

Effects provided by the program execution apparatus 1

A "method whereby the performance is greatly affected by transferring the program execution in the middle of the processing" that is optimized by the program execution apparatus 1 is, for example, a method that includes an endless loop, or a method wherein multiple iterations of a loop are performed that produce a difference in the priorities existing between the execution of the interpreter and the execution of the compiled code.

The code must be scanned before the execution in order to examine, without program execution, whether a loop is included. However, when there is a great amount of code, the time required for the scanning may cause the performance to be deteriorated.

While executing the program, the program execution apparatus 1 can detect a loop without scanning being required, and can transfer the interpreter to the compiled code to enable fast processing.

Generally, at an arbitrary point the interpreter is transferred to the compiled code, and in situations where transfers should be performed, in most cases transfer points are present in loops. However, since it is difficult to optimize a loop in which there is a transfer point, the processing for moving the transfer point outside the loop is an important performance enhancing procedure.

Advantages of the Invention

As is described above, according to the program execution method of the invention, even when a transfer point is present in a program to be changed, higher optimization extending across the transfer point can be performed.

In addition, according to the program execution method of this invention, in a multithreaded environment, even when an interpreter is transferred to compiled code at a plurality of transfer points, a plurality of compiled code processes are not required, and generated codes do not overlap.

What is claimed is: